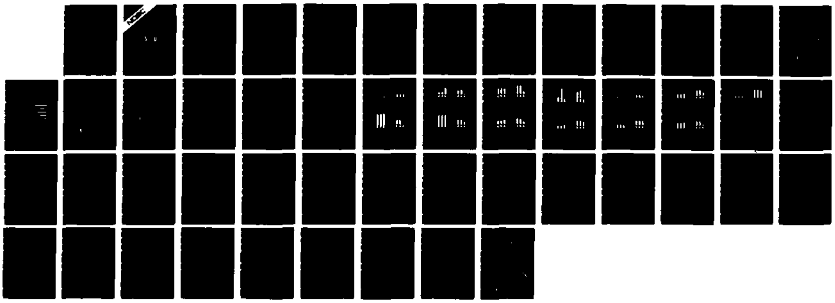
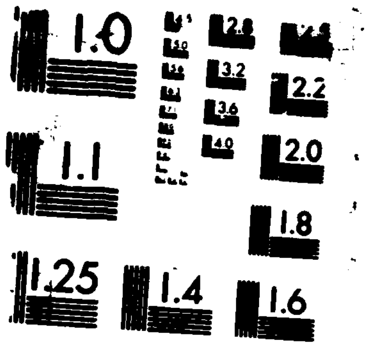
AD-A188 585 OPS83: STYLE GUIDE FOR HIGH PERFORMANCE(U) NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA L E GADBOIS JUN 87 NOSC/TR-1187 1/1

UNCLASSIFIED F/G 12/9 NL





MICROCOPY RESOLUTION TEST CHART

NOSC

NAVAL OCEAN SYSTEMS CENTER San Diego, California 92152-5000

Technical Report 1187
June 1987

OPS83: Style Guide for High Performance

L. E. Gadbois

Approved for public release; distribution is unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NOSC TR 1187	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Ocean Systems Center 6b OFFICE SYMBOL (if applicable) Code 444	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City State and ZIP Code) San Diego, CA 92152	7b ADDRESS (City State and ZIP Code)
8a NAME OF FUNDING SPONSORING ORGANIZATION Office of the Chief of Naval Research 8b OFFICE SYMBOL (if applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City State and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	AGENCY ACCESSION NO
IR Funding			

11 TITLE (Include Security Classification)
OPS83: Style Guide for High Performance

12 PERSONAL AUTHOR(S)
L.E. Gadbois

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM Oct 86 TO Jun 87	June 1987	51

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial intelligence, rule-based systems, OPS83, Rete algorithm, program optimisation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The Rete algorithm is employed in most of the shells that use a rule-based knowledge representation paradigm, as well as in other shells that provide procedural encoding. The Rete match algorithm performs comparisons between the items in working (data) memory and the patterns of the rule's antecedents. The size and configuration of both the working memory and the patterns markedly impact match times. Interactive effects of six parameters on both make and remove times are measured: number of rules; number of conditions per rule; number of comparisons per condition; total number of working-memory elements; number of working-memory elements matched with conditions; constant-condition, within-condition, and between-condition comparisons. (Keywords; artificial intelligence; program optimization).

20 DISTRIBUTION AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED UNLIMITED [✓ SAME AS RPT] ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Laurence E. Gadbois	22b TELEPHONE (Include Area Code) (619)225-7247 22c OFFICE SYMBOL Code 444

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

EXECUTIVE SUMMARY

OBJECTIVE

Examine the performance of the Rete algorithm of OPS83 for different working-memory element (WME) and rule left-hand side configurations. Results of this study can be used in designing OPS83 and similar language programs for maximum performance speed.

RESULTS

- Adding working-memory elements is generally faster than removing that same working-memory element. The larger the make time, the larger the relative difference between make and remove times. When it takes less than about 10 ms to make a WME, it takes 3 to 4 times as long to remove that same WME. Make times of around 25 ms correspond to remove times about 10 times as long. Make times of 300 to 400 ms correspond to remove times about 50 times as long.
- The number of rules has a linear impact on the time it takes to modify working memory.
- For most conditions, it takes about the same amount of time to match constant-condition as within-condition comparisons.
- Between-condition comparisons fluctuate from faster to slower than constant-condition and within-condition comparisons.
- The number of conditions has an exponential impact on make and remove times.
- The number of comparisons per condition has an erratic effect on the modification times, but is usually small in size.
- The number of working-memory elements that match conditions has an exponential impact on the modification times.
- The total number of working-memory elements affects the modification times. The magnitude is variable, as are the relative effects on the constant-, within-, and between-condition comparisons.
- Working-memory elements that match the first conditions of rules may take 3 to 4 times longer to modify than those which match the later parts of the LHS.

CONTENTS

FIGURES

TABLES

INTRODUCTION

DEVELOPMENT OF OPS83

The Rete algorithm was developed by Charles L. Forgy at Carnegie-Mellon University. It is widely accepted as an efficient pattern-matching algorithm. DARPA (Defense Advanced Research Project Agency) acknowledged the need for a high-execution-speed production system language by funding the development of OPS83. OPS83 uses the Rete algorithm and the C programming language to achieve high execution speed after compilation.

EXPERIENCE WITH OPS83

A large expert system using OPS83 and developed at NOSC and Carnegie-Mellon University has shown serious run-time performance limitations. Degradation appeared after extended run times or subjection to heavy loads. Extensive processing usually resulted in extensive working-memory sizes. The quest for means to improve performance speed has led to the in-depth measurements of the OPS83 implementation of the Rete algorithm reported here.

RETE MATCH ALGORITHM

Function/Overview

The function of the Rete match algorithm is to compute the conflict set, which is a set of ordered pairs of the form: [production, list of working-memory elements matched by the LHS]. A production is a rule in a production system, and the LHS (left-hand side) is the antecedent (or pattern of conditions) needed for the rule to be satisfied. These ordered pairs are called instantiations.

The main subtask of computing the conflict set is pattern matching, which involves performing comparisons between patterns specified in rule antecedents and items in working (data) memory. An antecedent describes a condition on a set of objects. These objects are items in the working memory and represent the problem-solving state of the system. The patterns may be the antecedents of the rules. The patterns are used to retrieve a set of objects that satisfy an antecendent. If a set of objects in working memory match an antecedent, that rule can be called with these objects bound to variables that can be used by the right-hand side of the rule.

The pattern matcher is called after each rule firing (execution). The right-hand sides can change working memory, so the sets of objects that match patterns can change too.

Why the Algorithm is Efficient

Comparing all the working-memory elements with all the condition comparisons becomes complex in that, every time a rule fires (executes), changes to the working memory may result. Before the next rule fires, the conflict set must be remade, which means that the match algorithm must be re-executed. Iteratively comparing all condition comparisons to the entire working memory after each rule firing is very expensive. The Rete match algorithm has incorporated methods to keep track of the matches from the previous rule firing. It therefore only needs to examine the newest change to working memory and decide: a) if any of the previous matches are affected by the change, and b) if any new matches result from the change.

The Rete algorithm exploits the fact that only a small fraction of working memory changes each cycle by storing the match from previous cycles in a network of queues called the Rete network and using them in subsequent cycles. It also exploits the similarity between condition elements of productions by performing common tests only once. These two features, in combination, make Rete an efficient match algorithm.

Data Flow Net

The Rete algorithm compiles the LHS of the rules into a data flow network to facilitate the pattern matching. To generate the network for a production, it begins with the individual condition elements in the left-hand side. For each condition element, it chains together test nodes that check:

— Whether the attributes in the condition element that have a constant as their value are satisfied.

— Whether the attributes in the condition element that are related to a constant by a predicate are satisfied.

— Whether two occurrences of the same variable within the condition element are consistently bound. Each node in the chain performs one such test. (The three kinds of tests above will be called "within-condition comparisons," because they correspond to individual condition elements and a single working-memory element.) Once the algorithm has finished with individual condition elements, it adds nodes that check for consistency of variable bindings across the multiple condition elements on the left-hand side. (These tests will be called "between-condition comparisons," because they refer to multiple condition elements and multiple working-memory elements.) Finally, the algorithm adds a special terminal node to represent the production corresponding to this part of the network.

The following discussion of the four kinds of nodes, and the tokens passed between them, is taken largely from Gupta, 1986. The objects that are passed between nodes are called "tokens," which consist of a tag and an ordered list of working-memory elements. The tag can be either a (+), indicating an addition to working memory, or a (-), indicating that something has been removed from working memory. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements on the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. These are as follows:

1) Constant-test nodes. These are used to test whether the attributes in the condition element that have a constant value are satisfied. These nodes always appear in the top part of the network. They have only one input and, as a result, they are sometimes called one-input nodes.

2) Memory nodes. These store the results of the match phase from previous cycles as states within them. The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. At a more detailed level, there are two types of memory nodes; (1) the alpha-mem nodes, and (2) the beta-mem nodes. The alpha-mem nodes store tokens that match individual comparisons. Thus all memory nodes immediately below constant-test nodes are alpha-mem nodes. The beta-mem nodes store tokens that match a sequence of condition elements in the left-hand side of a production. Thus all memory nodes immediately below two-input nodes are beta-mem nodes.

3) Two-input nodes. These test for joint satisfaction of condition element in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives at the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives at the right input of a two-input node. There are also two types of two-input nodes: (1) and-nodes and (2) not-nodes. While the the and-nodes are responsible for the positive condition elements and behave in the way described above, the not-nodes are responsible for the negated condition elements and behave in an opposite manner. The not-nodes generate a successor token only if there are no matching tokens in the memory node corresponding to the negated condition element.

4) Terminal nodes. There is one such node associated with each production in the program. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

State Saving

The saving of the match state between rule firings is time-saving for most applications. The Rete algorithm lies along a gradation of schemes that save different amounts of state between rule firings. The following is a discussion of what some of these states are, and where the Rete algorithm fits among them.

One possible scheme that a state-saving algorithm may use is to store information only about matches between individual condition elements and working-memory elements. In the terminology of the Rete algorithm, this means that only the state associated with the alpha-mem nodes is saved. For example, consider a production with three condition elements CE1, CE2, and CE3. Then the algorithm stores information about all working-memory elements that match CE1, all working-memory elements that match CE2, and all working-memory elements that match CE3. It does not, however, store working-memory tuples that satisify CE1 and CE2 together, and so on. The information is recomputed on each cycle. This scheme is at the low end of the state-saving algorithms. The problem with this scheme is that much of the state has to be recomputed on each cycle. This often increases the total time taken by the cycle.

A second possible scheme that a state-saving algorithm may use is to store information about matches between all possible combinations of condition elements that occur in the left-hand side of a production and the sequences of working-memory elements that satisfy them. For example, consider a production with three condition elements CE1, CE2, and CE3. In this scheme, the algorithm stores information about all working-memory elements that match CE1, CE2, and CE3 individually, as well as information about all working-memory tuples that match CE1 and CE2 together, CE2 and CE3 together, and so on. This scheme stands at the high end of the state-saving algorithms. It stores almost all information known about the matches between the productions and the working-memory. Two possible problems with the scheme are: 1) the state may become very large, and 2) the algorithm may spend a lot of time computing and deleting state that never gets used. That is, the state may never result in a production entering or leaving the conflict set.

The amount of state computed by the Rete algorithm falls in between that computed by the previous two schemes. The Rete algorithm stores information about working-memory elements that match individual condition elements, as proposed in the first scheme. In addition, it also stores information about tuples of working-memory elements that match some fixed combinations of condition elements. The choice about the combinations of condition elements for which match information is stored is fixed at the compiler's compile time. (Note that by varying the combinations of condition elements for which match information is stored, a large family of different Rete algorithms can be generated.) For example, for a production with three condit'on elements CE1, CE2, and CE3, the standard Rete algorithm stores information about working-memory elements that match CE1, CE2, and CE3 individually. This information is stored in the alpha-mem nodes. In addition, it stores information about working-memory element tuples that match CE1 and CE2 together. This information is stored in a beta-mem node. The Rete algorithm uses this information and combines it with the information about working-memory elements that match CE3 to generate tuples that match the complete left-hand side (CE1, CE2, and CE3 all together). The Rete algorithm does not store information about working-memory tuples that match CE1 and CE3 together, or those tuples that match CE2 and CE3 together, or those tuples that match CE2 and CE3 together, as is done by the algorithm in the second scheme.

The Rete algorithm exploits the similarity between condition elements of productions by sharing constant-test nodes, memory nodes, and two-input nodes. The sharing of nodes in the Rete network results in considerable savings in execution time, since the nodes have to be evaluated only once instead of multiple times. This sharing of nodes also results in savings in space. This is because sharing reduces the size of the Rete network and because sharing can collapse replicated tokens in multiple unshared memory nodes into a single token in a shared memory node.

The Rete algorithm is used in OPS83 and other places to manage the matching of the working-memory elements to the LHS conditions of the rules. This step has been shown to be one of the slowest phases in execution of OPS83 code. It is thus a ripe candidate for quantitative investigation to delineate where the slowest steps are. This will give pointers to software engineers and programmers concerned with high-speed performance.

APPROACH OF THE CURRENT REPORT

This report documents iterations of the various parameters that affect the steps taken by the match algorithm. Since the match algorithm compares working-memory elements with rule patterns, different sizes and configurations of these two parameters are examined.

METHODS

HARDWARE

Measurements were made on a SUN 3/160 workstation with 8 Mbytes of main memory (Motorola 68020 cpu). A 385K disk was used, with 150K set aside for swap space.

SOFTWARE

Version 2.1(B) of the OPS83 compiler was used for all measurements.

The UNIX "prof" program was used to measure the total run times. This program can be used to measure the percentage of time spent doing each routine. The sum of all these routines gives the amount of time used to run the whole program.

OPS83 provides a function call named "wmctime" that returns the amount of time used to conduct the last working-memory change. This call is used to measure both additions to and deletions from working memory.

The experiments presented in this paper outline how long it took to trace various working-memory changes through the net of LHS conditions compiled into the program. Each time working memory changes, the pattern matching between this change and the LHSs is performed. The wmctime function call returns the amount of time it took to make the working-memory modifications and to perform the pattern matching. Comparison of varying working-memory and rule configurations shows the features of rule set structures that are slow and those that are fast under different, varying working-memory configurations.

The software setup was a combination of a UNIX shell program (appendix A) that passed parameters to several OPS83 programs (appendixes B, C, and D) that used these parameters to write other OPS83 programs (appendix E). The shell program then compiled and ran these OPS83 programs, capturing the results of the wmctime calls. The parameters passed by the UNIX shell program were the number of rules, the number of condition elements per rule, the number of comparisons within each condition element, and the size of working memory to use. The UNIX program iterated over a set of combinations of the parameters. These sets of parameters were passed to the three OPS83 programs in appendixes B, C, and D, which differed in whether the comparisons were constant, within, or between conditions. These programs then wrote the final test programs, from which the results presented here were obtained.

Care had to be exercised in the use of the wmctime function. As mentioned above, it gave the amount of time used in making a wm modification. As other computer processes competing for the cpu would interrupt the wm modification being measured, the output of the wmctime would need to have compensated for the time that it was not active on the cpu. Tests were conducted to determine whether, in fact, computing jobs were considered. It turned out that major competing jobs (compiling and running other programs, editing, etc.) were compensated for, but monitoring the motion of the mouse was not. To prevent biases in results, all the tests were run as the only job on the system.

Wmctime returned the time used in increments of 20 milliseconds of cpu time. This is a high relative error when the actual time is only several milliseconds. To compensate for this, each test was run 30 times; the average time for the 30 runs is reported in the results. To show the variability returned by the wmctime function, the raw data for tables 1 and 2 are presented in appendix F.

RESULTS

Experiment #1: The results of this experiment are fundamental to the design of all the subsequent experiments.

Five-part hypothesis: First, there is a difference between the amount of time it takes to make and the amount of time it takes to remove a WME. Second, the time it takes to make or remove a WME depends on whether it is the first WME or the second, third, *et cetera.* Third, the number of rules in the program affect the make/remove time. Fourth, the number of conditions per rule affect the make/remove time. Fifth, unbiased variance in the measurement technique may dictate that the average of many runs could be a better estimate of actual speed.

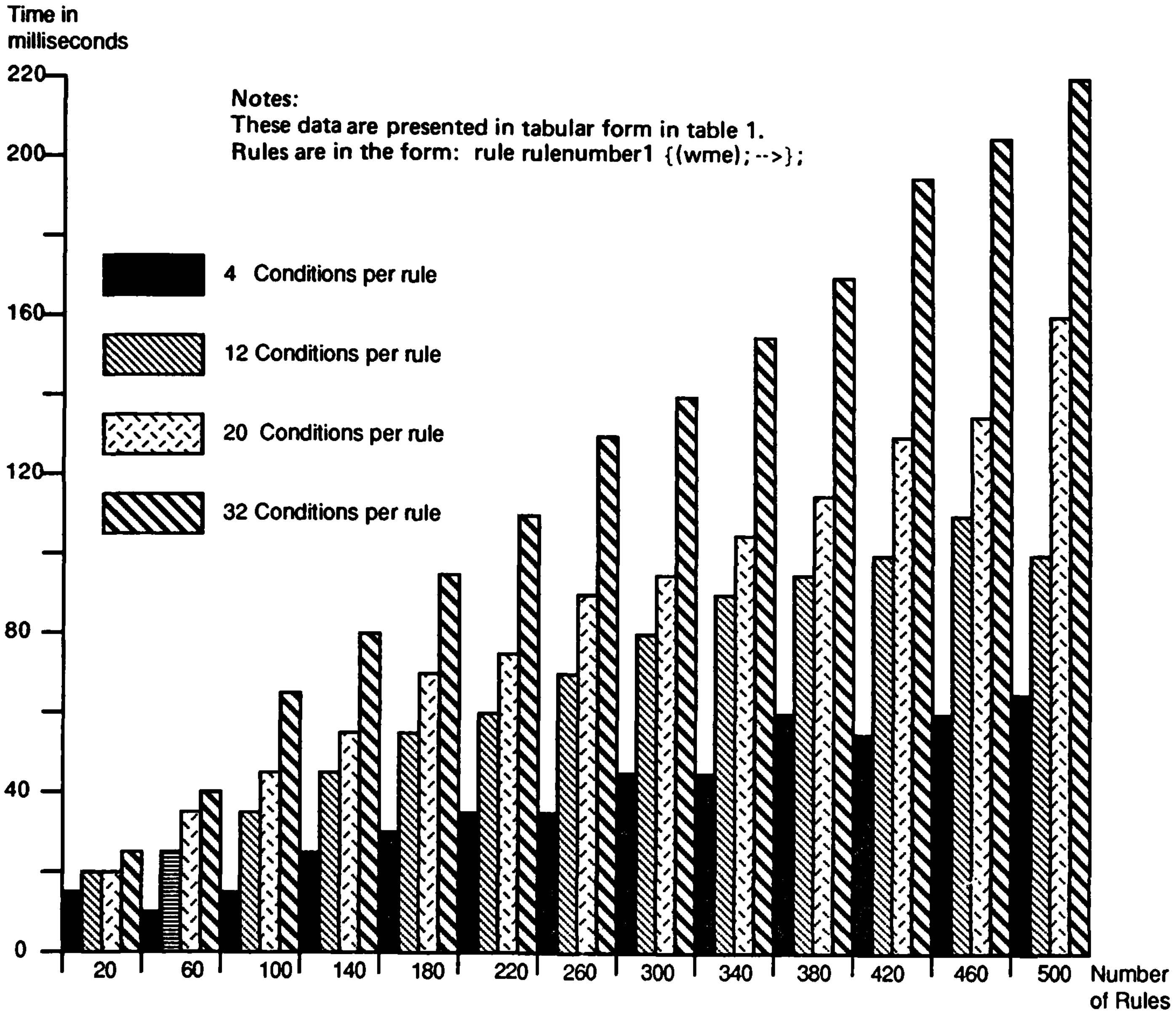
Results (see figures 1–4, supported by tables 1–4):

- It takes many times longer to remove than to make a WME for large rule sets. The larger the rule set, the longer the relative difference between make and remove times.
- It takes longer to add and remove a second WME than the first.
- The more rules in the program, the greater the modification time.
- The more matching conditions per rule, the greater the modification time.
- The measurement technique returns time in 20-millisecond increments, and there is a small variance in the results.

Significance: Since all these parameters impact speed, they need to be quantified in all subsequent experiments. Comparison of any test results must check for consistency in these parameters or factor their impact into the results achieved.

Significance: Since these parameters impact speed, they are likely candidate parameters to measure in greater detail for bottlenecks to avoid in programming.

Significance: For experiments where the make/remove time is small, the error is relatively large. The raw data presented in appendix F show the variance obtained. All results presented in this report (except appendix F) are the average of 30 replicates.



Notes:
These data are presented in tabular form in table 1.
Rules are in the form: rule rulenumber1 {(wme); -->};

Figure 1. Effect of the number of conditions and rules on the time to make the first working-memory element.

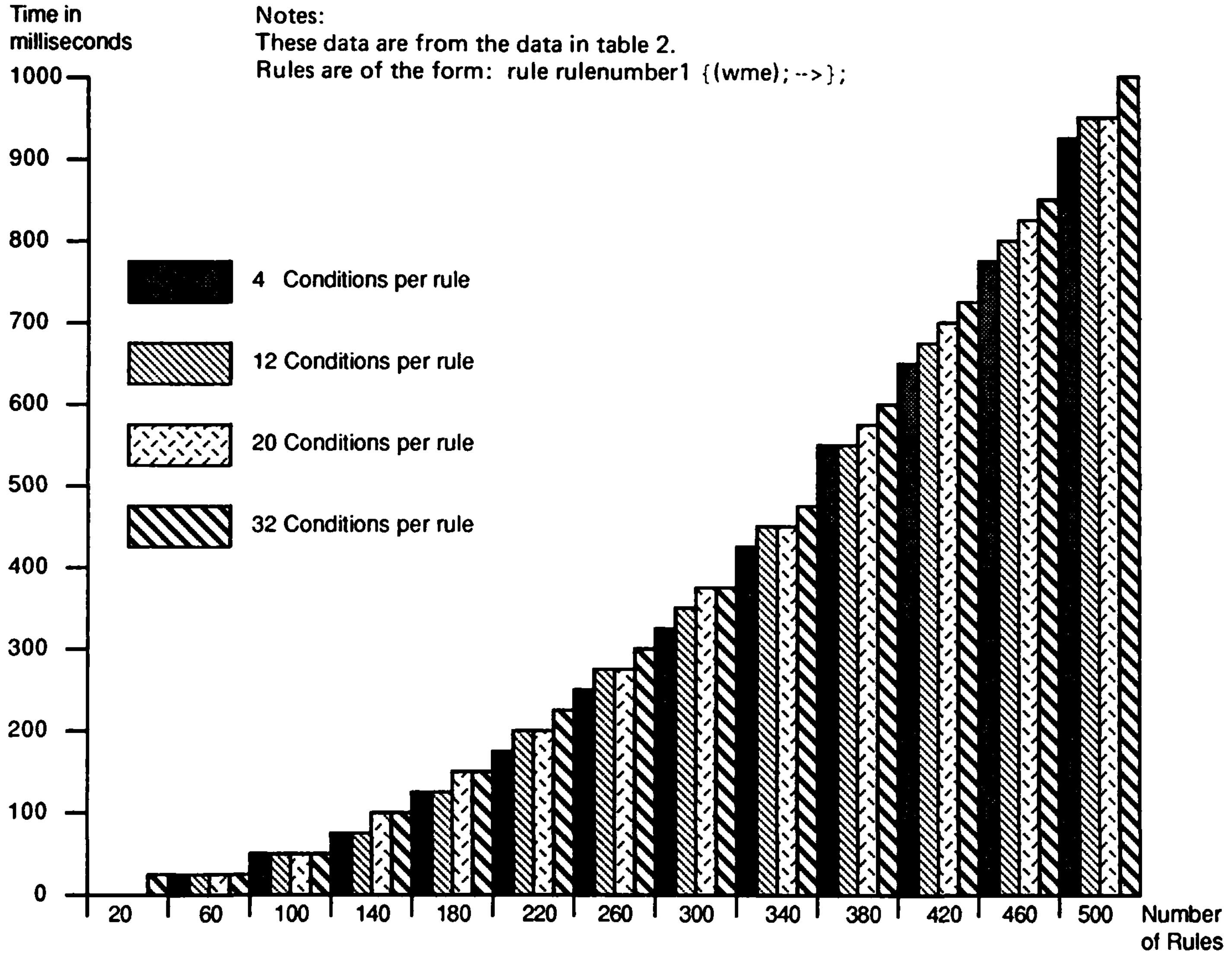
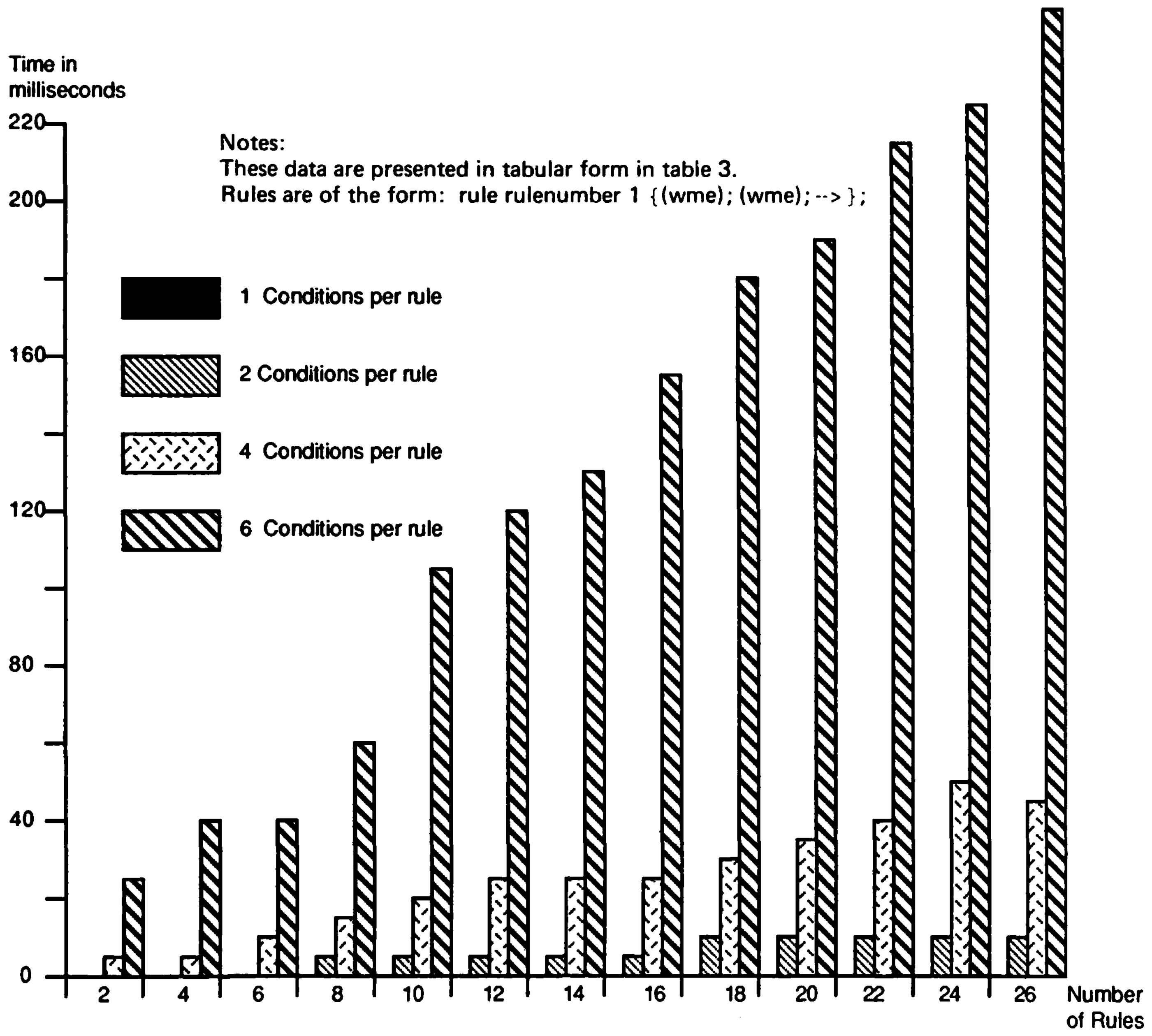


Figure 2. Effect of the number of antecedents and rules on the time to remove the first working-memory element.



Notes:
These data are presented in tabular form in table 3.
Rules are of the form: rule rulenumber 1 {(wme); (wme); --> };

Figure 3. Effect of the number of conditions and rules on the time to make the second working-memory element.

Notes:
These data are presented in tabular form in table 4.
Rules are of the form: rule rulenumber1 {(wme); (wme); --> };

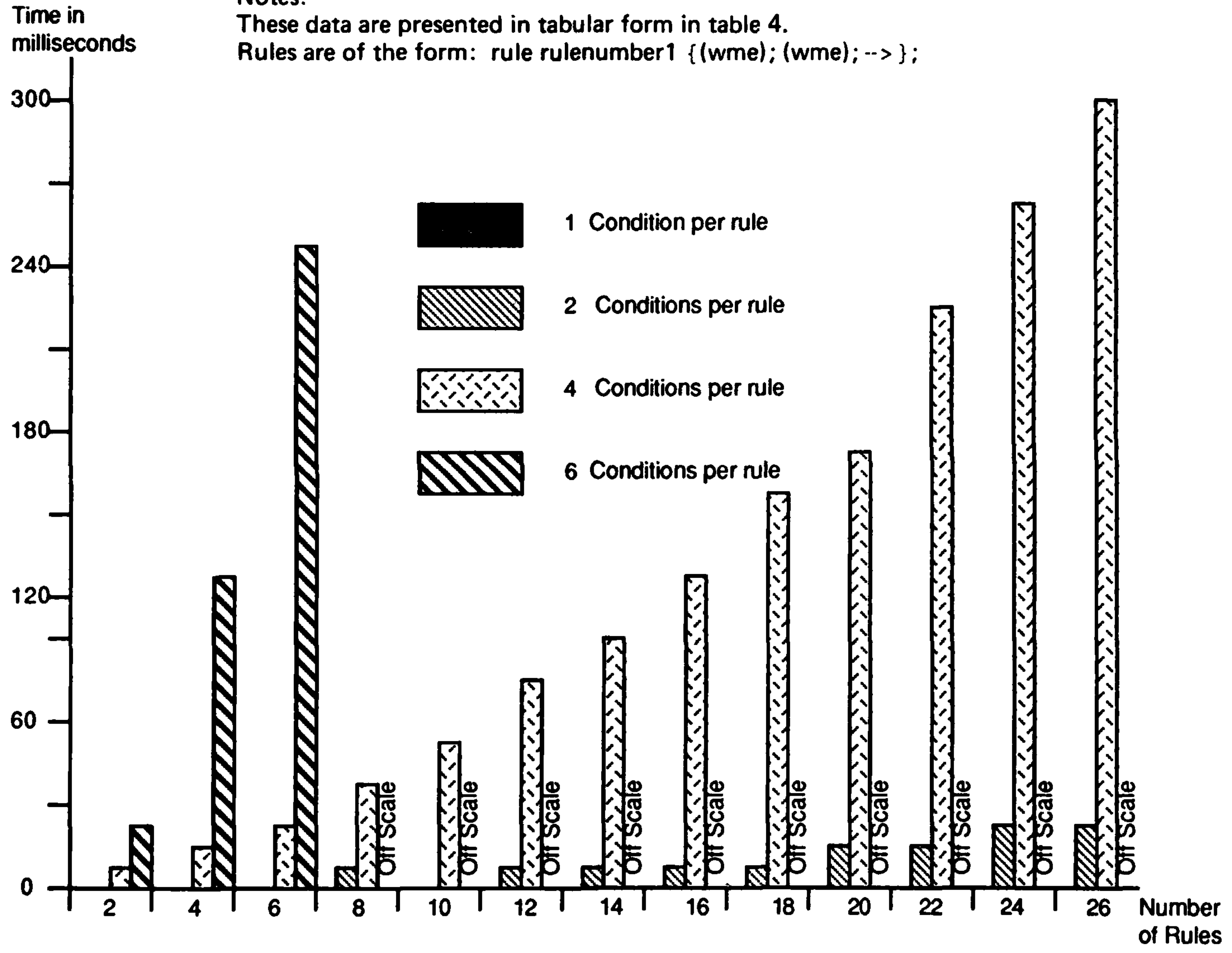


Figure 4. Effect of the number of conditions and rules on the time to remove the second working-memory element.

Table 1. Average time (30 trials) in milliseconds to make the first working-memory element.

	Number of Conditions per Rule			
Number of Rules	4	12	20	32
20	12.67	19.33	20.00	27.33
60	12.00	26.00	32.67	41.33
100	17.33	36.00	46.67	64.67
140	26.67	43.33	54.00	80.00
180	30.00	54.00	69.33	95.33
220	35.33	61.33	77.33	110.67
260	35.33	71.33	89.33	130.67
300	43.33	78.00	95.33	142.00
340	43.33	88.00	104.00	155.33
380	59.33	94.00	117.33	170.00
420	54.67	100.00	132.00	197.33
460	59.33	111.33	136.00	207.33
500	67.33	110.67	158.00	218.67

Table 2. Average time (30 trials) in milliseconds to remove the first working-memory element.

	Number of Conditions per Rule			
Number of Rules	4	12	20	32
20	4.67	4.00	6.67	14.00
60	18.00	21.33	25.33	29.33
100	42.00	48.67	55.33	62.00
140	76.67	85.33	88.00	103.33
180	128.00	135.33	145.33	154.00
220	184.67	198.00	206.00	224.00
260	260.00	270.00	280.00	298.67
300	336.67	352.00	368.00	386.00
340	435.33	445.33	460.00	486.00
380	538.00	552.67	570.67	594.67
420	650.00	674.00	690.00	716.67
460	784.00	798.00	822.00	847.33
500	921.33	941.33	940.67	998.00

Table 3. Average time (30 trials) in milliseconds to make the second working-memory element.

	Number of Conditions per Rule			
Number of Rules	1	2	4	6
2	1.33	0.00	5.33	22.67
4	0.00	2.00	7.33	42.00
6	1.33	2.00	7.33	42.00
8	1.33	2.67	12.67	59.33
10	1.33	2.67	18.00	103.33
12	0.67	4.00	23.33	118.00
14	0.67	7.00	23.33	134.67
16	0.00	3.33	26.00	153.33
18	2.00	8.67	30.00	182.00
20	1.33	9.33	33.33	186.00
22	2.00	9.33	40.00	216.67
24	0.67	10.00	48.67	222.67
26	0.33	8.67	43.33	250.67

Table 4. Average time (30 trials) in milliseconds to remove the second working-memory element.

	Number of Conditions per Rule			
Number of Rules	1	2	4	6
2	0.00	0.67	4.67	26.67
4	1.33	0.67	11.33	115.33
6	2.00	1.33	21.33	230.00
8	0.67	4.00	34.00	392.00
10	0.67	1.33	50.67	592.67
12	1.33	5.33	72.00	836.67
14	2.00	9.33	91.33	1120.67
16	0.33	8.67	116.00	1454.67
18	0.67	10.00	143.33	1823.33
20	1.33	13.33	168.67	2217.33
22	0.67	16.67	208.00	2686.00
24	2.67	20.00	244.00	3188.67
26	4.00	22.67	284.00	3712.00

MAKE, REMOVE, AND TOTAL RUN TIMES

Under nearly trivial conditions of very small working-memory sizes and a small number of very simple rules, remove time is smaller than make time. Under all other conditions, however, remove time is larger than make time—often dramatically so. Under continuous running conditions, an OPS83 program will need to remove as much data from working memory as it adds. Since the amount of time it takes to perform removes is larger than that required by makes, the remove time will have a greater impact on the overall speed. The discussions in this paper present both make times and remove times, but they will be biased toward remove times since total run times exhibit this same bias.

Generally, when make times are small, remove times are also fairly small (figures 5a-b). As make times increase slightly, remove times increase markedly (figures 6a-b). As make times increase even more, remove times explode, reaching the computer's limits (figures 7a-b).

Total run times were also measured to determine how much of the actual run time was spent in the pattern matching. The experimental setups used for the results in this report were geared to specifically focus on the Rete algorithm match time. No time was spent firing rules, doing conflict resolution, *et cetera*. As a result, more than 99% of the total run time was spent doing pattern matching. A few selected results are presented to confirm this.

For figures 8a-b, the total run time for constant comparisons was 53.38 seconds. The amount of time spent making and removing the 30 replicate working-memory elements was 53.26 seconds. Thus 0.12 second was spent doing OPS83 overhead and performing the first two makes to get up to experimental conditions.

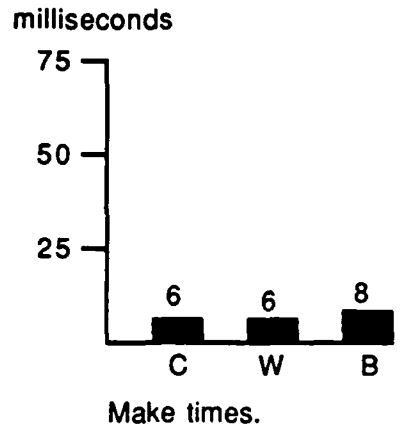
CONSTANT-, WITHIN-, AND BETWEEN-CONDITION COMPARISONS

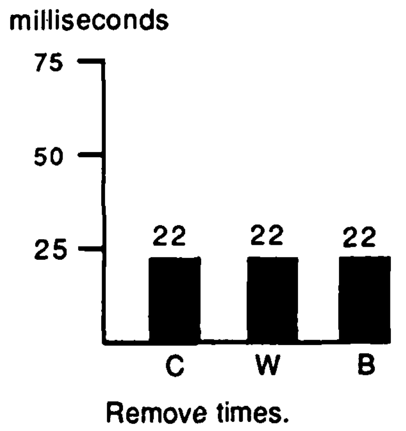
Figures 5a through 17b show the amount of time needed to do constant-, within-, and between-condition comparisons under a suite of rule and working-memory configurations. Constant- and within-condition comparisons resulted in nearly identical times to modify working memory. A marked exception to this is evident in figures 11a-b. Both make and remove times were much larger for within-condition comparisons than constant-condition comparisons. (For a complete guide to figure and table comparisons, see table 5.)

By comparing figures 11a with 16a and figures 11b with 16b, it is evident that, by reducing the numer of comparisons per condition, the disparity in time to perform constant- and within-condition comparisons is removed.

By comparing figures 11a with 14a and figures 11b with 14b, it is evident that, by reducing the total number of working-memory elements, the disparity in time to perform constant- and within-condition comparisons is removed.

The pattern of how much time it takes to perform between-condition comparisons is variable relative to constant- and within-condition comparisons. Remove times may be slower (figures 6b, 8b to 11b) or about the same (figures 5b, 7b, 12b to 17b) as the other two types. Make time may be faster (figure 11a) or slower (figure 7a), or about the same (figures 8a, 9a, 10a, 12a, 13a, 5a, 14a, 6a, 15a, 16a, 17a) as the other two types.

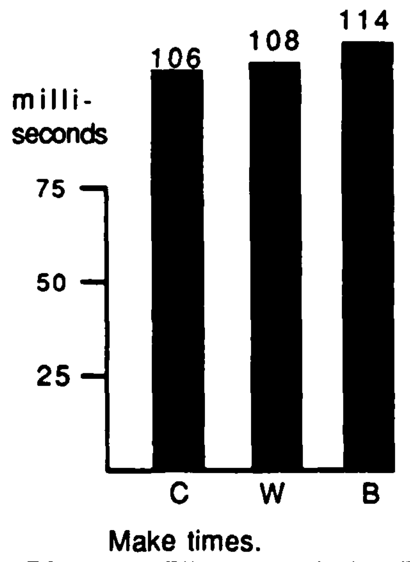


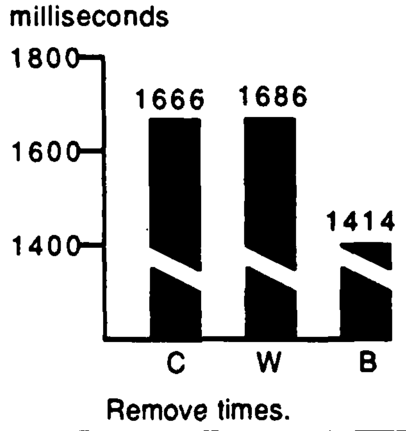


Notes:
No. of wmes = 4
No. of wmes matching all conditions = 4
No. of rules = 2
No. of conditions = 3
No. of comparisons/condition = 4

C, W, and B stand for constant-, within-, and between-condition comparisons.

Figure 5a-b. Make/remove times for few conditions and few rules.

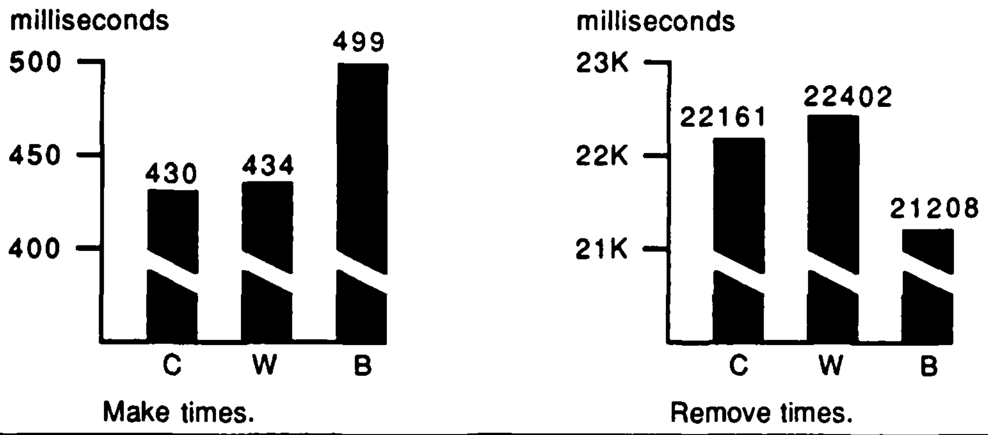




Notes:
No. of wmes = 8
No. of wmes matching all conditions = 2
No. of rules = 8
No. of conditions = 5
No. of comparisons/condition = 2

C, W, and B stanc for constant-, within-, and between-condition comparisons.

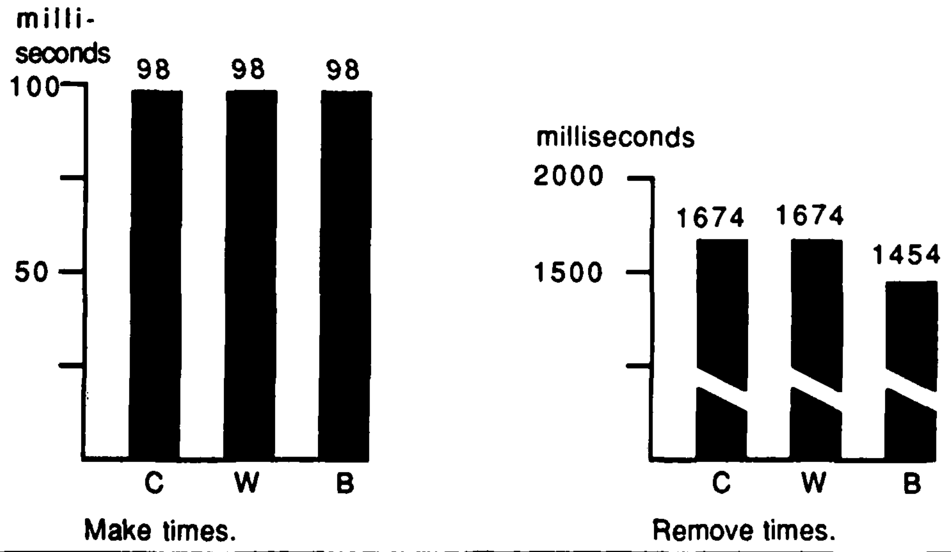
Figure 6a-b. Make/remove times for few matching working-memory elements and comparisons per condition.



Notes:
No. of wmes = 3
No. of wmes matching all conditions = 2
No. of rules = 8
No. of conditions = 5
No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

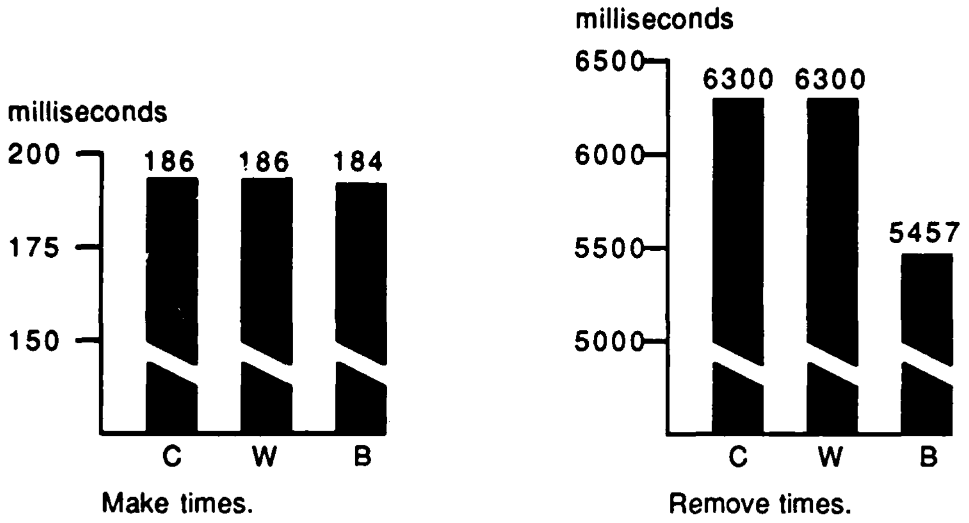
Figure 7a-b. Make/remove times with a larger working memory.



Notes:
No. of wmes = 3
No. of wmes matching all conditions = 3
No. of rules = 4
No. of conditions = 5
No. comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

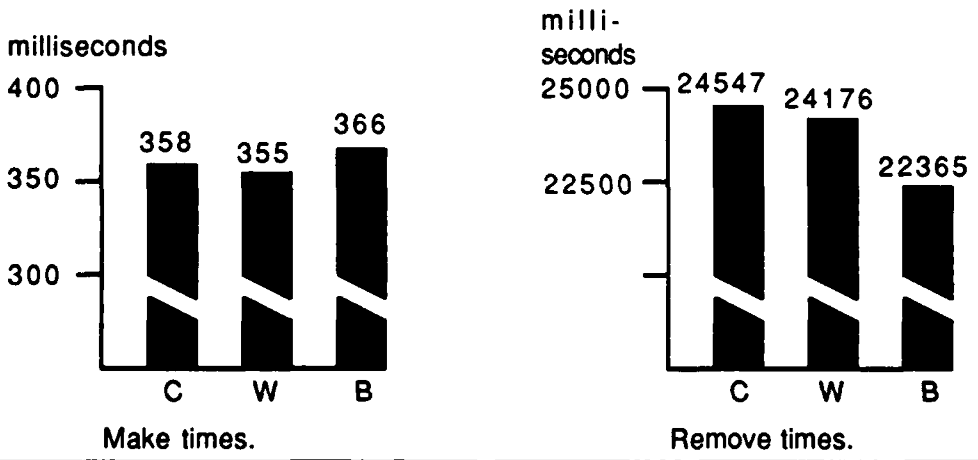
Figure 8a-b. Make/remove times for few rules and working-memory elements.



Notes:	No. of rules = 8
No. of wmes = 3	No. of conditions = 5
No. of wmes matching all conditions = 3	No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

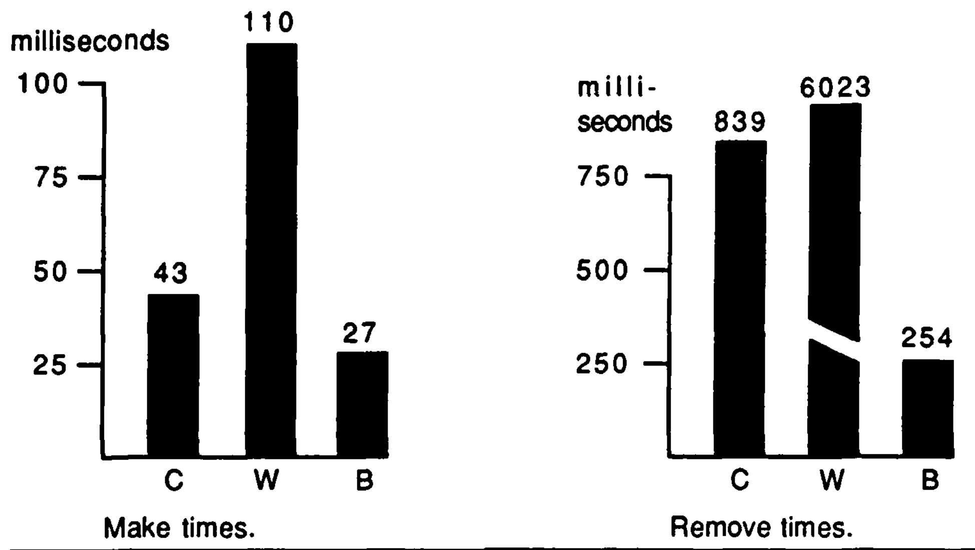
Figure 9a-b. Make/remove times for many simple conditions.



Notes:	No. of rules = 2
No. of wmes = 8	No. of condtions = 5
No. of wmes matching all conditions = 4	No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

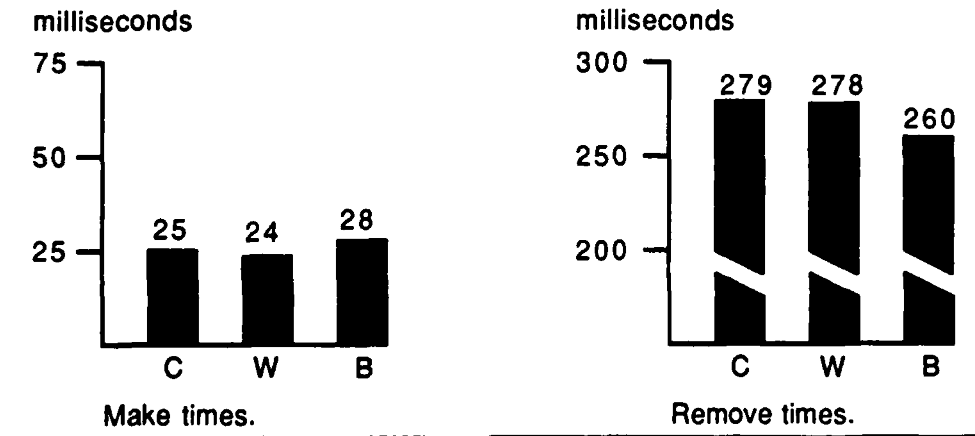
Figure 10a-b. Make/remove times for few rules with many conditions.



Notes:	No. of rules = 8
No. of wmes = 8	No. of condition = 3
No. of wmes matching all conditions = 4	No. of comparisons/condition = 4

C, W, and B stand for constant-, within-, and between-condition comparisons.

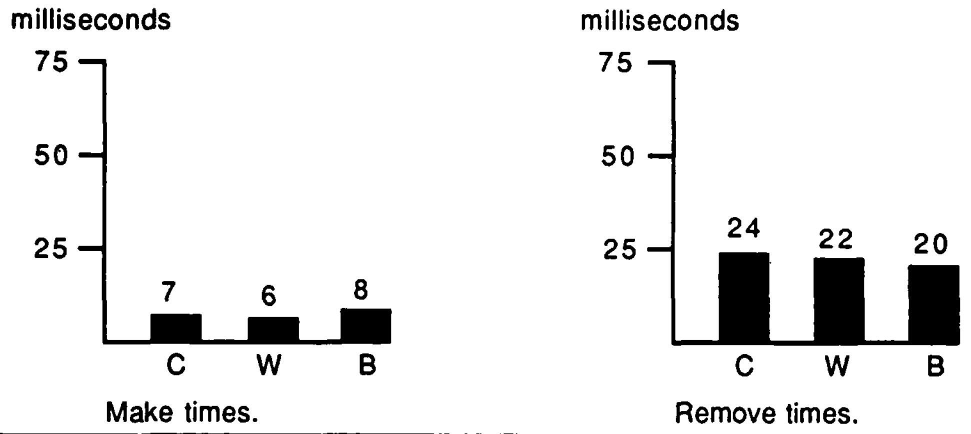
Figure 11a-b. Make/remove times with disparity in constant- and within-condition comparisons.



Notes:	No. of rules = 8
No. of wmes = 4	No. of conditions = 3
No. of wmes matching all conditions = 4	No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

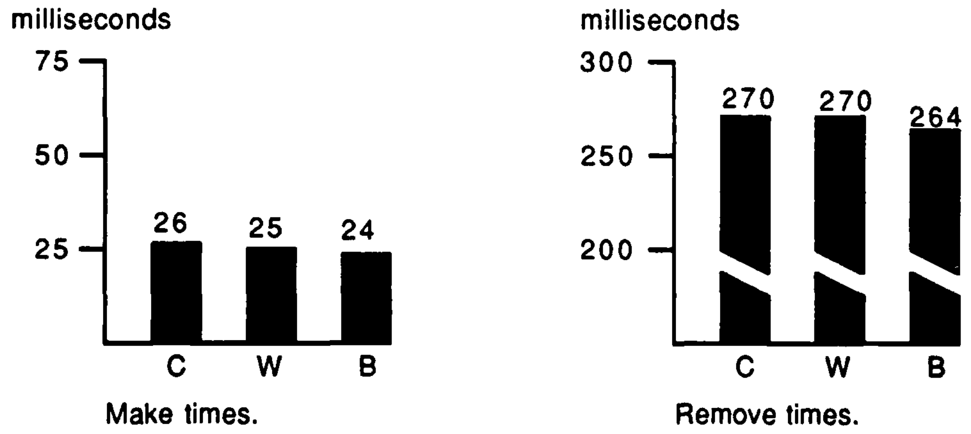
Figure 12a-b. Make/remove times for few simple conditions.



Notes:	No. of rules = 2
No. of wmes = 4	No. of conditions = 3
No. of wmes matching all conditions = 4	No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

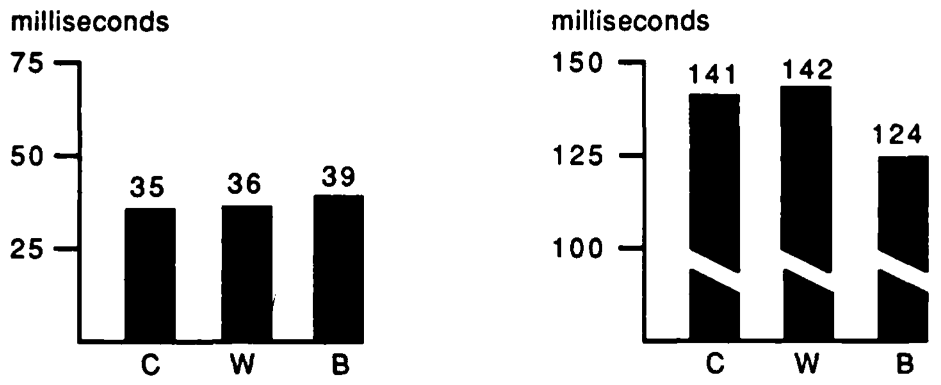
Figure 13a-b. Make/remove times for a small simple rule set and working-memory.



Notes:	No. of rules = 8
No. of wmes = 4	No. of conditions = 3
No. of wmes matching all conditions = 4	No. of comparisons/condition = 4

C, W, and B stand for constant-, within-, and between-condition comparisons.

Figure 14a-b. Make/remove times for few conditions.

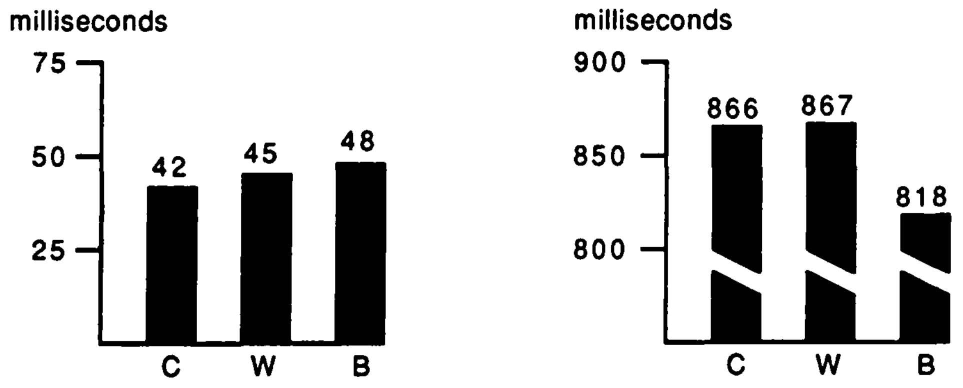


Make times. Remove times.

Notes:
No. of wmes = 8
No. of wmes matching all conditions = 2
No. of rules = 2
No. of conditions = 5
No. comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

Figure 15a-b. Make/remove times for few rules and matching working-memory elements.

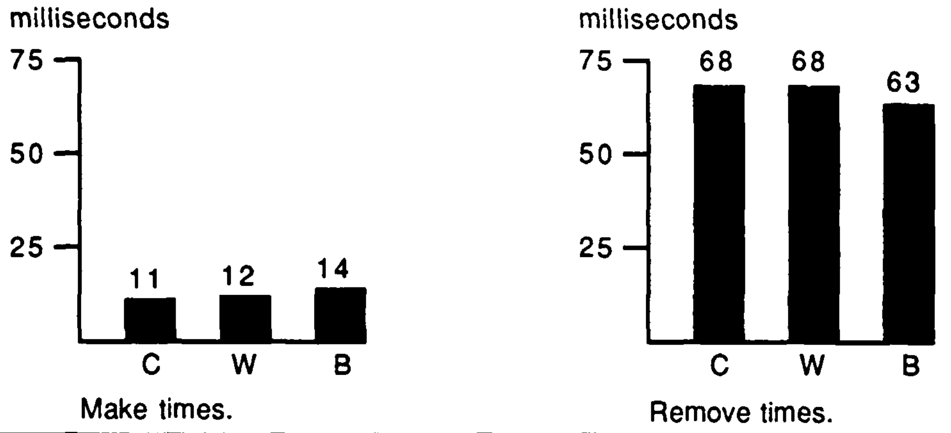


Make times. Remove times.

Notes:
No. of wmes = 8
No. of wmes matching all conditions = 4
No. of rules = 8
No. of conditions = 3
No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

Figure 16a-b. Make/remove times for few conditions and comparisons per condition.



Notes:	No. of rules = 2
No. of wmes = 8	No. of conditions = 3
No. of wmes matching all conditions = 4	No. of comparisons/condition = 2

C, W, and B stand for constant-, within-, and between-condition comparisons.

Figure 17a-b. Make/remove times for few rules with simple left-hand sides.

Table 5. Guide to figure and table comparisons.

Type of Comparison	Make Time	Remove Time
No. of rules	Tables 1, 3	Tables 2, 4
	Figures 1, 3	Figures 2, 4
	Figures 8a, 9a	Figures 8b, 9b
	Figures 12a, 13a	Figures 12b, 13b
	Figures 5a, 14a	Figures 5b, 14b
No. of conditions per rule	Figures 1, 3	Figures 2, 4
	Figures 10a, 17a	Figures 10b, 17b
No. of comparisons per condition	Figures 11a, 16a	Figures 11b, 16b
	Figures 12a, 14a	Figures 12b, 14b
	Figures 13a, 5a	Figures 13b, 5b
No. of WMEs matching all conditions	Figures 10a, 15a	Figures 10b, 15b
	Figures 7a, 9a	Figures 7b, 9b
No. of total working-memory elements	Figures 11a, 14a	Figures 11b, 14b
	Figures 7a, 6a	Figures 7b, 6b
	Figures 12a, 16a	Figures 12b, 16b

EFFECT OF THE NUMBER OF RULES

The more rules in the program, the greater the amount of time needed to trace a working-memory modification through the net. This applies for both make and remove times. Make times increase linearly with the number of rules, while remove times exhibit an exponential response to rule set size.

Make time doubles if the rule set size is doubled. For unrealistically simple rule LHSs, the effect of the number of rules is shown in figures 1 through 4 (these raw data are also presented in tables 1-4, and in appendix F). The same relationship is shown under other rule/data memory configurations. Figure 12a has four times the number of rules as figure 13a, and has make times about four times larger. Figure 9a has twice as many rules as figure 8a, and has make times about twice as large. Figures 5a and 14a show further support for this pattern.

Remove time increases nearly fourfold with a twofold increase in the number of rules. This can be seen by comparing figures 8b and 9b, 12b and 13b, and tables 2 and 4 (the same data are presented in figures 2 and 4).

EFFECT OF THE NUMBER OF CONDITIONS PER RULE

Both make and remove time increased dramatically as the number of conditions per rule increased. The explosion and the make time associated with increasing the number of conditions are shown in figures 1 and 3. This increase is attributable to the combinatorial explosion resulting from the cross products of conditions with working-memory elements.

Remove time also shows an exponential explosion in going from three to five conditions per rule (see figures 2 and 4).

EFFECT OF THE NUMBER OF COMPARISONS PER CONDITION

The number of comparisons per condition had an erratic impact on the make times. Comparison of figures 13a to 5a, and 12a to 14a, showed negligible effects from changing the number of comparisons per condition. Comparison of figure 11a to 16a shows no change in constant comparisons, but a difference in make times for within- and between-condition comparisons. The contributing difference between this pair of figures and the other two pairs is that, for this pair, not every WME matched every condition. In figures 11a and 16a, the condition tests prevented total matches between the conditions and some of the WMEs.

Remove times exhibited the same pattern as make times. Comparing figures 11b and 16b shows that, for the case in which not every WME matches every condition, increasing the number of comparisons per condition has no impact on constant comparisons, greatly increases the within-condition comparison times, and lowers the between-condition times. Comparing figure 14b to 12b, and 5b to 13b, shows no effect from changes in the number of comparisons.

EFFECT OF THE NUMBER OF WORKING-MEMORY ELEMENTS MATCHING ALL CONDITIONS

The number of working-memory elements that match the antecedent conditions has an exponential impact on the make and remove times (compare figures 7 to 9, and 10 to 15).

EFFECT OF THE NUMBER OF TOTAL WORKING-MEMORY ELEMENTS

Increasing the total number of working-memory elements increases the make and remove times. The magnitude is variable, as are the relative effects on the constant-, within-, and between-condition comparisons. Increasing the number of WMEs from three to eight gave about a fourfold increase for all three types of comparisons (compare figures 7a and 6a). Doubling the number of WMEs gave less than a doubling in make time for the conditions in figures 12a and 16a. For the conditions in figures 11a and 14a, a doubling of WMEs gave varied results. Constant comparisons had slightly less than a doubling in make time for a doubling in total WMEs. Between-condition comparions gave no significant effect. Within-condition comparisons took more than four times as long.

Remove times about trebled, with a doubling in the number of WMEs for the conditions shown in figures 12b and 16b. Comparing figures 11b and 14b results in a different response pattern. Doubling the number of WMEs gave more than a threefold increase in constant comparisons, a 22-fold increase in within-condition comparisons, and no significant change in between-condition comparisons.

EFFECT OF THE CONDITION'S POSITION IN THE LHS

Table 6 shows the match times for the individual condition elements on the LHS of a rule. Viturally identical results were achieved among the constant-, within-, and between-condition comparison types. As working-memory elements were added, they would match most quickly with the conditions at the beginning of the LHS, and take progressively longer for the conditions later in the LHS.

For the experimental conditions used, WMEs that match the first condition took much longer to remove than WMEs that match the last conditions. Adding the make and remove times for any condition position shown in table 6 indicates that it is slower to modify working-memory elements that match the first few conditions than those that match the last ones.

Table 6. Amount of time needed to match the different LHS condition elements.

[The rule set had 16 rules. Each rule had five LHS conditions. Each condition except the first had two comparisons. Three WMEs were made with all fields in the value array equal to 0. Each working-memory element matched exactly one LHS condition element. The make and remove times are for the third WME. Two WMEs of each type were made first, and then the third WME of each type was made in numerically increasing order (i.e., WME1, WME2. . .) and then were removed in decreasing order (i.e., WME80, WME79. . .).]

The rules were of the form:

```
rule
&1 (WME1)
&2 (WME2 value[1] = 0 value[2] = 0)
&3, &4, and &5 similar to &2

rule next
&1 (WME6)
&2 (WME7 value[1] = 0 value[2] = 0)
&3, &4, and &5 similar to &2 .
```

LHS position of matching:

Condition Element	Make Time	Remove Time
First	3.1	40.3
Second	4.9	22.2
Third	6.9	11.7
Fourth	9.1	7.2
Fifth	8.4	4.4

A similiar experiment was conducted by using within-condition comparisons. The rules were of the form:

```
rule
&1 (WME6)
&2 (WME7 value[1] = @.value[1] value[2] = @.value[2])
&3, &4, and &5 similar to &2 .
```

LHS position of matching:

Condition Element	Make Time	Remove Time
First	3.1	40.1
Second	4.8	21.5
Third	7.0	12.2
Fourth	8.8	6.3
Fifth	8.5	4.4

A similiar experiment was conducted by using within-condition comparisons. The rules were of the form:

```
rule
&1 (WME6)
&2 (WME7 value[1] = &1.value[1] value[2] = &1.value[2])
&3 (WME8 value[1] = &2.value[1] value[2] = &2.value[2])
&4 and &5 similar to &3 .
```

LHS position of matching:

Condition Element	Make Time	Remove Time
First	3.0	40.1
Second	5.8	22.6
Third	7.4	12.2
Fourth	8.8	7.1
Fifth	9.8	3.8

SUMMARY OF RESULTS

- Adding working-memory elements is generally faster than removing that same working-memory element. The larger the make time, the larger the relative difference between make and remove times. When it takes less than about 10 ms to make a WME, it takes 3 to 4 times as long to remove that same WME. Make times of around 25 ms correspond to remove times about 10 times as long. Make times of 300 to 400 ms correspond to remove times about 50 times as long.
- The number of rules has a linear impact on the time it takes to modify working memory.
- For most conditions, it takes about the same amount of time to match constant-condition as within-condition comparisons.
- Between-condition comparisons fluctuate from faster to slower than constant-condition and within-condition comparisons.
- The number of conditions has an exponential impact on make and remove times.
- The number of comparisons per condition had an erratic effect on the modification times, but was usually small in size.
- The number of working memory elements that match conditions has an exponential impact on the modification times.
- The total number of working-memory elements affects the modification times. The magnitude is variable, as are the relative effects on the constant-, within-, and between-condition comparisons.
- Working-memory elements that match the first conditions of rules may take 3 to 4 times longer to modify than those which match the later parts of the LHS.

CONCLUSIONS

The number of rules has an algebraic impact on the modification times, while the number of conditions has an exponential impact. Thus, it is better to have more rules with fewer conditions than vice versa.

The number of comparisons in a condition has little effect on match time. Thus the use of complex conditions, which are very selective about which WMEs to fire, is an efficient programming technique.

No sharp breaks in the modification times were seen as the parameters (no. of rules, no. of conditions per rule, no. of comparisons per condition, no. of working-memory elements matching all conditions, no. of total working-memory elements) used in this study were varied. Parameters that exerted an impact did not have threshold values below which changes had little impact but above which there was marked change. Change was continuous, which facilitated a predictable degradation or improvement in performance as these parameters changed.

Optimal program formats are highly dependent on the task. For a task where speed is critical, working-memory size must be kept as small as possible. Combining many small working-memory elements into fewer larger ones is more efficient. If the task is very short, working-memory management can be performed two ways. Because removes generally are slower than makes, an efficient alternative to removes for a short run may be to add working-memory elements that "poison" existing elements to keep rules from firing on them.

DISCUSSION

During the initial stages of building an expert system, efficiency lies in the reduction of programming time. Thus knowledge representation, working-memory element design, and other factors should be geared towards easing the burden of the programmer. Guidelines for optimal performance should be used so that blatant violations of good programming practice do not occur. After the building of the prototype for proof of concept, the program can be examined both holistically and fragment by fragment to determine potential sites for optimization.

There are rules of thumb to follow for efficient programming. A good summary of these rules is found in Brownston, et al. (1985), p 241. Conformance to such guidelines is a good first approach to an efficient program. If performance is still below requirements after adhesion to the guidelines, more extensive steps must be taken. At this point, tradeoffs in different aspects of program performance are balanced by using the detailed results presented in this report.

An additional consideration for the software engineer is whether the program will run on a continuous basis or simply run to find a single answer and then exit. The difference from the performance perspective is that a continuously running program must perform working-memory housekeeping to keep from being overloaded with data. Thus, following initial startup, there are roughly as many "additions to" as "deletions from" working memory. This means that remove times are an important consideration. For a program that will run only for a short time, remove time is not as important as make time.

Writing efficient OPS83 programs requires both a knowledge of how the Rete algorithm works and a knowledge of the problem space. The data input and the output desired have a strong bearing on the best knowledge and data representation. Lots of disjunct information necessitates lots of working-memory elements. Alternatively, the task and data could consist of lots of information, groups of which have a common trait. If this is the case, the software engineer has the option of combining some or all of the data fragments that have trait(s) in common in the same working-memory element. Rules that need to bring in large amounts of information on their LHS can do so more quickly by using several large working-memory elements than by using many small working-memory elements.

Multiple rules with identical LHS conditions do not have an exponential impact on modification time because the constant, memory, and two input nodes of the network for the rules are shared. Terminal nodes are unique for each rule, as is inclusion in the conflict set, so there is an impact on modfication time, but the combinatorial explosion seen when the number of conditions is increased does not occur with changes in the number of rules.

BIBLIOGRAPHY

Brownston, Lee, et al. An Introduction to Rule-Based Programming, Addison-Wesley, 1985.

Forgy, Charles. Rete: A Fast Algorithm for the Many Pattern, Many Object Pattern Match Problem, *Artificial Intelligence*, 19, 17-37, 1982.

Gupta, Anoop, and C. Forgy. Measurements on Production Systems, Carnegie-Mellon Univ. Pub. CMU-CS-83-167, 1983.

Gupta, Anoop. Parallelism in Production Systems: The Sources and the Expected Speed-up, Carnegie-Mellon Univ. Pub. CMU-CS-84-169, 1984.

Gupta, Anoop. Parallelism in Production Systems, Carnegie-Mellon Univ. Pub. CMU-CS-86-122, 1986.

APPENDIX A
UNIX SHELL SCRIPT THAT QUERIES THE USER FOR CONFIGURATION PARAMETERS

This is a UNIX shell program used to set the number of rules, the number of conditions per rule, the number of comparisons per condition, and the size of working memory to use in the experiment. It passes these parameters to three OPS83 programs that set up test programs to measure constant-, within-, and between-condition comparisons.

```
echo ""
ps -a       # Check for other jobs running
for rules in 2
do
for conditions in 4
do
 echo "     # Need a newline
 for comparisons in 2
 do
 for makes in 4 8
  do
   echo "$rules $conditions $comparisons $makes" | setup3run
   make -f irmake
    irrun
    prof irrun > junk
    tail -1 junk  # Displays the total run time
  done
 done
done
done
ps -a
for rules in 2
do
for conditions in 4
```

```
do
 echo "
 for comparisons in 2
 do
 for makes in 4 8
  do
   echo "$rules $conditions $comparisons $makes" | setup2run
    make -f irmake
     irrun
     prof irrun > junk
     tail -1 junk
  done
 done
done
done
ps -a
for rules in 2
do
for conditions in 4
do
 echo "
 for comparisons in 2
 do
 for makes in 4 8
  do
   echo "$rules $conditions $comparisons $makes" | setup1run
    make -f irmake
     irrun
     prof irrun > junk
     tail -1 junk
  done
 done
done
done
ps -a
```

APPENDIX B

CODE THAT CONFIGURES CONSTANT-CONDITION COMPARISON PROGRAMS

Example code that reads the number of rules, the number of conditions per rule, the number of comparisons per condition, and the total number of WMEs. This input is used to write a program to do constant-condition comparisons.

```
module setup(main1) {


use o83shl;


procedure main1() {
  local &l: logical, &LHS: integer, &rules: integer, &i: integer, &j: integer,
      &file: integer, &bar: char, &comps: integer, &k: integer,
      &makes: integer, &total: integer;
  &bar = '|';
  &file = create(|ir.ops|);
  if (&file <= 0) write() |Error creating ir.ops|, '\n'
else {
  write() '\n', |Of the format @.value[n] = 0|;
  write() '\n', |# of rules: |;
  read() &rules;
  write() &rules;
  write() '\n', |# of LHS conditions: |;
  read() &LHS;
  write() &LHS;
  write() '\n', |# of comparisons per LHS condition: |;
  read() &comps;
  write() &comps;
  write() '\n', |# of makes: |;
  read() &makes;
  write() &makes, '\n';
  write(&file) |module ir1(main1) {|, '\n',
           |use o83shl;|, '\n',
           |-- This program written by setup1.ops|, '\n',
           |external function wmctime(): integer;|, '\n';
write(&file) |type WME = element (value:array(17: integer) );|, '\n';

for &i = (1 to &rules) {
 write(&file) |rule number|, &i, | { |;
 write(&file) '\n', | &0(WME);|;
 for &j = (1 to &LHS) {
  write(&file) '\n', | &|, &j, |(WME|;
-- To build: (WME (@.value[1] = 0); (@.value[2] = 0); );
  for &k = (1 to &comps-1)
    write(&file) |  value[|, &k, |] = 0;|;
    write(&file) |  value[|, &k, |] = 0;|;
  write(&file) |); |;
 }; -- end for &j = (1 to &LHS)
 write(&file) |--> };|, '\n';
}; -- end for &i = 1 to &rules

write(&file) |procedure main1() {|, '\n',
           |  local &i: integer, &j: integer, &k: integer,|, '\n';
write(&file) |       &make_WME_cum : integer,|,
           | &remove_WME_cum : integer,|, '\n';
write(&file) |       &l: logical;|, '\n';
write(&file) |&make_WME_cum = 0;  |,
           | &remove_WME_cum = 0;|, '\n';
for &i = (1 to (&makes - 4))
  write(&file) |make (WME value[2] = 1);|, '\n';
  write(&file) |make (WME value[2] = 0);|, '\n';
  write(&file) |make (WME value[2] = 0);|, '\n';
  write(&file) |make (WME value[2] = 0);|, '\n';
write(&file) |for &i = (1 to 30) { -- Thirty reps.|, '\n';
  write(&file) |  make (WME);|, '\n',
                | &make_WME_cum = &make_WME_cum + wmctime();|, '\n';
    write(&file) |  &k = wsize();|, |  &l = wremove(&k);|, '\n',
         |  if (&l = 0b)|,
         |    write() '\n', |, &bar, |****ERROR IN REMOVE****|,
         &bar, '\n',
         |  else |, '\n',
                |    &remove_WME_cum = &remove_WME_cum + wmctime();|, '\n';
    write(&file) |}; -- end for &i = 1 to 30|, '\n';
    write(&file) |   write() '\n', |, &bar, |Avg WME make/remove time: |,
         &bar, |,|, '\n', |        &make_WME_cum/30, |, &bar, |  |, &bar,
         |, &remove_WME_cum/30, '\n';|, '\n';
    write(&file) |}; -- end procedure main1|, '\n',
                |}; -- end module|, '\n';
}; -- end else
&l = close(&file);
}; -- end procedure main1

}; -- end module setup
```

APPENDIX C
CODE THAT CONFIGURES WITHIN-CONDITION COMPARISON PROGRAMS

Example code that reads the number of rules, the number of conditions per rule, the number of comparisons per condition, and the total number of WMEs. This input is used to write a program to do within-condition comparisons.

```
module setup(main1) {

use o83shl;

procedure main1() {
  local &l: logical, &LHS: integer, &rules: integer, &i: integer, &j: integer,
       &file: integer, &bar: char, &comps: integer, &k: integer,
       &makes: integer, &total: integer;
  &bar = '|';
  &file = create(|ir.ops|);
  if (&file <= 0) write() |Error creating ir.ops|, '\n'
else {
  write() '\n', |Of the format @.value[n] = @.value[n]|;
  write() '\n', |# of rules: |;
  read() &rules;
  write() &rules;
  write() '\n', |# of LHS conditions: |;
  read() &LHS;
  write() &LHS;
  write() '\n', |# of comparisons per LHS condition: |;
  read() &comps;
  write() &comps;
  write() '\n', |# of makes: |;
  read() &makes;
  write() &makes, '\n';
  write(&file) |module ir1(main1) {|, '\n',
            |use o83shl;|, '\n',
            |-- This program written by setup2.ops|, '\n',
            |external function wmctime(): integer;|, '\n';
  write(&file) |type WME = element (value:array(17: integer) );|, '\n';

  for &i = (1 to &rules) {
   write(&file) |rule number|, &i, | { |;
   write(&file) '\n', | &0(WME);|;
   for &j = (1 to &LHS) {
    write(&file) '\n', | &|, &j, |(WME|;
-- To build: (WME (@.value[1] = @.value[1]); (@.value[2] = @.value[1]); );
    for &k = (1 to &comps-1)
      write(&file) |  value[|, &k, |] = @.value[|, &k, |];|;
      write(&file) |  value[|, &k, |] = @.value[|, &k-1, |];|;
    write(&file) |); |;
   }; -- end for &j = (1 to &LHS)
   write(&file) |--> };|, '\n';
  }; -- end for &i = 1 to &rules

  write(&file) |procedure main1() {|, '\n',
            | local &i: integer, &j: integer, &k: integer,|, '\n';
  write(&file) |       &make_WME_cum : integer,|,
            | &remove_WME_cum : integer,|, '\n';
  write(&file) |       &l: logical;|, '\n';
  write(&file) |&make_WME_cum = 0;  |,
            | &remove_WME_cum = 0;|, '\n';
  for &i = (1 to (&makes - 4))
    write(&file) |make (WME value[2] = 1);|, '\n';
    write(&file) |make (WME value[2] = 0);|, '\n';
    write(&file) |make (WME value[2] = 0);|, '\n';
    write(&file) |make (WME value[2] = 0);|, '\n';
  write(&file) |for &i = (1 to 30) { -- Thirty reps.|, '\n';
    write(&file) |  make (WME);|, '\n',
                | &make_WME_cum = &make_WME_cum + wmctime();|, '\n';
    write(&file) |   &k = wsize();|, |  &l = wremove(&k);|, '\n',
        |  if (&l = 0b)|,
        |    write() '\n', |, &bar, |****ERROR IN REMOVE****|,
        &bar, '\n',
        |  else |, '\n',
                |    &remove_WME_cum = &remove_WME_cum + wmctime();|, '\n';
    write(&file) |}; -- end for &i = 1 to 30|, '\n';
    write(&file) |   write() '\n', |, &bar, |Avg WME make/remove time: |,
        &bar, |,|, '\n', |        &make_WME_cum/30, |, &bar, |  |, &bar,
        |, &remove_WME_cum/30, '\n';|, '\n';
    write(&file) |}; -- end procedure main1|, '\n',
                |}; -- end module|, '\n';
}; -- end else
&l = close(&file);
}; -- end procedure main1

}; -- end module setup
```

APPENDIX D

CODE THAT CONFIGURES BETWEEN-CONDITION COMPARISON PROGRAMS

Example code that reads the number of rules, the number of conditions per rule, the number of comparisons per condition, and the total number of WMEs. This input is used to write a program to do between-condition comparisons.

```
module setup(main1) {

use o83shl;

procedure main1() {
 local &l: logical, &LHS: integer, &rules: integer, &i: integer, &j: integer,
     &file: integer, &bar: char, &comps: integer, &k: integer,
     &makes: integer, &total: integer;
 &bar = '|';
 &file = create(|ir.ops|);
 if (&file <= 0) write() |Error creating ir.ops|, '\n'
else {
 write() '\n', |Of the format @.value[n] = @-1.value[n]|;
 write() '\n', |# of rules: |;
 read() &rules;
 write() &rules;
 write() '\n', |# of LHS conditions: |;
 read() &LHS;
 write() &LHS;
 write() '\n', |# of comparisons per LHS condition: |;
 read() &comps;
 write() &comps;
 write() '\n', |# of makes: |;
 read() &makes;
 write() &makes, '\n';
 write(&file) |module ir1(main1) {|, '\n',
            |use o83shl;|, '\n',
            |-- This program written by setup3.ops|, '\n',
            |external function wmctime(): integer;|, '\n';
write(&file) |type WME = element (value:array(17: integer) );|, '\n';

for &i = (1 to &rules) {
  write(&file) |rule number|, &i, | { |;
  write(&file) '\n', | &0(WME);|;
  for &j = (1 to &LHS) {
    write(&file) '\n', | &|, &j, |(WME|;
-- To build: (WME (@.value[1] = @.value[1]); (@.value[2] = @-1.value[1]); );
    for &k = (1 to &comps-1)
        write(&file) |  value[|, &k, |] = &|, &j-1, |.value[|, &k, |];|;
        write(&file) |  value[|, &k, |] = &|, &j-1, |.value[|, &k-1, |];|;
    write(&file) |); |;
  }; -- end for &j = (1 to &LHS)
  write(&file) |--> };|, '\n';
}; -- end for &i = 1 to &rules

write(&file) |procedure main1() {|, '\n',
            | local &i: integer, &j: integer, &k: integer,|, '\n',
write(&file) |        &make_WME_cum : integer,|,
            | &remove_WME_cum : integer,|, '\n';
write(&file) |        &l: logical;|, '\n';
write(&file) |&make_WME_cum = 0;  |,
            | &remove_WME_cum = 0;|, '\n';
for &i = (1 to (&makes - 4))
   write(&file) |make (WME value[2] = 1);|, '\n';
   write(&file) |make (WME value[2] = 0);|, '\n';
   write(&file) |make (WME value[2] = 0);|, '\n';
   write(&file) |make (WME value[2] = 0);|, '\n';
write(&file) |for &i = (1 to 30) { -- Thirty reps.|, '\n';
   write(&file) |   make (WME);|, '\n',
                |  &make_WME_cum = &make_WME_cum + wmctime();|, '\n';
        write(&file) |  &k = wsize();|, |  &l = wremove(&k);|, '\n',
            |  if (&l = 0b)|,
            |    write() '\n', |, &bar, |****ERROR IN REMOVE****|,
            &bar, '\n',
            |  else |, '\n',
                |    &remove_WME_cum = &remove_WME_cum + wmctime();|, '\n';
        write(&file) |}; -- end for &i = 1 to 30|, '\n';
        write(&file) |  write() '\n', |, &bar, |Avg WME make/remove time: |,
            &bar, |,|, '\n', |        &make_WME_cum/30, |, &bar, |  |, &bar,
            |, &remove_WME_cum/30, '\n';|, '\n';
        write(&file) |}; -- end procedure main1|, '\n',
                |}; -- end module|, '\n';
    }; -- end else
    &l = close(&file);
    }; -- end procedure main1

    }; -- end module setup
```

APPENDIX E

EXAMPLE CODE FOR BETWEEN-CONDITION COMPARISONS

Example code for two rules, five conditions per rule, two comparisons per condition, four working-memory elements, and four working-memory elements matching all conditions. This is the code that would result when the numbers {2, 4, 2, 4} are passed to the program in appendix E.

```
module ir1(main1) {
use o83shl;
-- This program written by setup3.ops
external function wmctime(): integer;
type WME = element (value:array(17: integer) );
rule number1 {
 &0(WME);
 &1(WME  value[1] = &0.value[1];  value[2] = &0.value[1];);
 &2(WME  value[1] = &1.value[1];  value[2] = &1.value[1];);
 &3(WME  value[1] = &2.value[1];  value[2] = &2.value[1];);
 &4(WME  value[1] = &3.value[1];  value[2] = &3.value[1];); --> };
rule number2 {
 &0(WME);
 &1(WME  value[1] = &0.value[1];  value[2] = &0.value[1];);
 &2(WME  value[1] = &1.value[1];  value[2] = &1.value[1];);
 &3(WME  value[1] = &2.value[1];  value[2] = &2.value[1];);
 &4(WME  value[1] = &3.value[1];  value[2] = &3.value[1];); --> };
procedure main1() {
 local &i: integer, &j: integer, &k: integer,
     &make_WME_cum : integer, &remove_WME_cum : integer,
     &l: logical;
&make_WME_cum = 0;  &remove_WME_cum = 0;
make (WME value[2] = 0);
make (WME value[2] = 0);
make (WME value[2] = 0);
for &i = (1 to 30) { -- Thirty reps.
  make (WME);
  &make_WME_cum = &make_WME_cum + wmctime();
  &k = wsize();  &l = wremove(&k);
  if (&l = 0b)     write() '\n', |****ERROR IN REMOVE****|
  else
   &remove_WME_cum = &remove_WME_cum + wmctime();
}; -- end for &i = 1 to 30
  write() '\n', |Avg WME make/remove time: |,
     &make_WME_cum/30, |  |, &remove_WME_cum/30, '\n';
}; -- end procedure main1
}; -- end module
```

APPENDIX F
DATA FOR TABLES 1 AND 2

Time is in milliseconds to make and remove the first WME.

Rules,	Conditions,	Type of Modification,	Number of Trials@Time				
20	4	make	11@0	19@20			
		remove	23@0	7@20			
20	12	make	9@0	13@20	8@40		
		remove	24@0	6@20			
20	20	make	10@0	10@20	10@40		
		remove	20@0	10@20			
20	32	make	2@0	17@20	9@40	2@60	
		remove	9@0	21@20			
60	4	make	12@0	18@20			
		remove	3@0	27@20			
60	12	make	4@0	14@20	11@40	1@60	
		remove	28@20	2@40			
60	20	make	3@0	11@20	10@40	6@60	
		remove	22@20	8@40			
60	32	make	8@20	12@40	10@60		
		remove	17@20	12@40	1@60		
100	4	make	6@0	22@20	2@40		
		remove	1@20	25@40	4@60		
100	12	make	1@0	11@20	11@40	7@60	
		remove	17@40	13@60			
100	20	make	1@20	19@40	9@60	1@80	
		remove	7@40	23@60			
100	32	make	3@20	4@40	8@60	13@80	2@100
		remove	27@60	3@80			
140	4	make	4@0	12@20	14@40		
		remove	5@60	25@80			
140	12	make	1@0	4@20	14@40	11@60	
		remove	22@80	8@100			

Rules,	Conditions,	Type of Modification,	Number of Trials@Time				
140	20	make	5@20	7@40	10@60	8@80	
		remove	18@80	12@100			
140	32	make	2@40	6@60	12@80	10@100	
		remove	25@100	5@120			
180	4	make	2@0	12@20	15@40	1@60	
		remove	19@120	10@140	1@160		
180	12	make	4@20	6@40	16@60	3@80	1@100
		remove	9@120	19@140	2@160		
180	20	make	1@0	1@20	1@40	13@60	8@80
			6@100				
		remove	22@140	8@160			
180	32	make	13@80	11@100	6@120		
		remove	9@140	21@160			
220	4	make	1@0	14@20	6@40	9@60	
		remove	1@160	21@180	8@200		
220	12	make	8@40	14@60	6@80	2@100	
		remove	6@180	21@200	3@220		
220	20	make	3@40	9@60	9@80	7@100	2@120
		remove	21@200	9@220			
220	32	make	6@80	9@100	9@120	5@140	1@160
		remove	1@200	22@220	7@240		
260	4	make	6@0	6@20	9@40	7@60	2@80
		remove	4@240	22@260	4@280		
260	12	make	5@40	10@60	8@80	7@100	
		remove	15@260	15@280			
260	20	make	2@40	2@60	10@80	12@100	4@120
		remove	4@260	22@280	4@300		
260	32	make	2@80	3@100	10@120	7@140	8@160
		remove	4@280	24@300	2@320		
300	4	make	10@20	7@40	11@60	2@80	
		remove	6@320	23@340	1@360		
300	12	make	1@40	10@60	11@80	7@100	1@120
		remove	13@340	16@360	1@380		
300	20	make	2@60	8@80	16@100	3@120	1@140

Rules,	Conditions,	Type of Modification,	Number of Trials@Time				
		remove	19@360	10@380	1@400		
300	32	make	4@100	5@120	10@140	6@160	5@180
		remove	22@380	7@400	1@420		
340	4	make	1@0	5@20	12@40	12@60	
		remove	10@420	17@440	3@460		
340	12	make	4@60	13@80	10@100	3@120	
		remove	1@420	20@440	9@460		
340	20	make	1@60	11@80	6@100	6@120	5@140
			1@160				
		remove	4@440	22@460	4@480		
340	32	make	3@100	1@120	4@140	14@160	8@180
		remove	21@480	9@500			
380	4	make	1@20	6@40	16@60	7@80	
		remove	8@520	17@540	5@560		
380	12	make	1@40	9@80	17@100	3@120	
		remove	11@540	19@560			
380	20	make	1@60	2@80	9@100	9@120	6@140
			3@160				
		remove	14@560	16@580			
380	32	make	4@120	3@140	7@160	9@180	4@200
			3@220				
		remove	11@580	16@600	3@620		
420	4	make	1@20	7@40	21@60	1@80	
		remove	17@640	11@660	2@680		
420	12	make	2@60	10@80	8@100	6@120	4@140
		remove	11@660	17@680	2@700		
420	20	make	4@80	5@100	3@120	9@140	5@160
			4@180				
		remove	17@680	12@700	1@740		
420	32	make	2@140	1@160	6@180	13@200	6@220
			2@240				
		remove	10@700	15@720	5@740		
460	4	make	2@20	10@40	8@60	7@80	3@100
		remove	4@760	19@780	4@800	3@820	

Rules,	Conditions,	Type of Modification,	Number of Trials@Time				
460	12	make	1@60	14@100	11@120	4@140	
		remove	4@780	25@800	1@820		
460	20	make	2@100	12@120	10@140	4@160	2@200
		remove	4@800	19@820	7@840		
460	32	make	1@140	2@160	8@180	5@200	7@220
			2@240	5@260			
		remove	19@840	11@860			
500	4	make	5@40	13@60	8@80	4@100	
		remove	2@900	25@920	2@940	1@960	
500	12	make	7@80	7@100	11@120	4@140	1@180
		remove	1@820	1@920	23@940	2@960	2@980
			1@1000				
500	20	make	1@100	4@120	6@140	7@160	10@180
			2@200				
		remove	29@940	1@960			
500	32	make	5@180	7@200	5@220	11@240	2@260
		remove	1@960	7@980	20@1000	1@1040	1@1080

END
DATE
FILMED
3-88
DTIC